

Heuristics in Conflict Resolution

Christian Drescher and Martin Gebser and Benjamin Kaufmann and Torsten Schaub

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract

Modern solvers for Boolean Satisfiability (SAT) and Answer Set Programming (ASP) are based on sophisticated Boolean constraint solving techniques. In both areas, conflict-driven learning and related techniques constitute key features whose application is enabled by conflict analysis. Although various conflict analysis schemes have been proposed, implemented, and studied both theoretically and practically in the SAT area, the heuristic aspects involved in conflict analysis have not yet received much attention. Assuming a fixed conflict analysis scheme, we address the open question of how to identify “good” reasons for conflicts, and we investigate several heuristics for conflict analysis in ASP solving. To our knowledge, a systematic study like ours has not yet been performed in the SAT area, thus, it might be beneficial for both the field of ASP as well as the one of SAT solving.

Introduction

The popularity of Answer Set Programming (ASP; (Baral 2003)) as a paradigm for knowledge representation and reasoning is mainly due to two factors: first, its rich modeling language and, second, the availability of high-performance ASP systems. In fact, modern ASP solvers, such as *clasp* (Gebser *et al.* 2007a), *cmodels* (Giunchiglia, Lierler, & Maratea 2006), and *smodels_{cc}* (Ward & Schlipf 2004), have meanwhile closed the gap to Boolean Satisfiability (SAT; (Mitchell 2005)) solvers. In both fields, conflict-driven learning and related techniques have led to significant performance boosts (Bayardo & Schrag 1997; Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001; Gebser *et al.* 2007d). The basic prerequisite for the application of such techniques is *conflict analysis*, that is, the extraction of non-trivial reasons for dead ends encountered during search. Even though ASP and SAT solvers exploit different inference patterns, their underlying search techniques are closely related to each other. For instance, the basic search strategy of SAT solver *chaff* (Moskewicz *et al.* 2001), nowadays a quasi standard in SAT solving, is also exploited by ASP solver *clasp*, in particular, the principles of conflict analysis are similar. Vice versa, the solution enumeration approach implemented in *clasp* (Gebser *et al.* 2007b) could also be applied by SAT solvers. Given these similarities, general search or, more specifically, conflict analysis techniques developed in one community can

(almost) immediately be exploited in the other field too.

In this paper, we address the problem of identifying “good” reasons for conflicts to be recorded within an ASP solver. In fact, conflict-driven learning exhibits several degrees of freedom. For instance, several constraints may become violated simultaneously, in which case one can choose the conflict(s) to be analyzed. Furthermore, distinct schemes may be used for conflict analysis, such as the resolution-based First-UIP and Last-UIP scheme (Zhang *et al.* 2001). Finally, if conflict analysis is based on resolution, several constraints may be suitable resolvents, likewise permitting to eliminate some literal in a resolution step.

For the feasibility of our study, it was necessary to prune dimensions of freedom in favor of predominant options. In the SAT area, the *First-UIP scheme* (Marques-Silva & Sakallah 1999) has empirically been shown to yield better performance than other known conflict resolution strategies (Zhang *et al.* 2001). We thus fix the conflict analysis strategy to conflict resolution according to the First-UIP scheme. Furthermore, it seems reasonable to analyze the first conflict detected by a solver (although conflicts encountered later on may actually yield “better” reasons). This leaves to us the choice of the resolvents to be used for conflict resolution, and we investigate this issue with respect to different goals: reducing the size of reasons to be recorded, skipping greater portions of the search space by backjumping (explained below), reducing the number of conflict resolution steps, and reducing the overall number of encountered conflicts (roughly corresponding to runtime). To this end, we modified the conflict analysis procedure of our ASP solver *clasp*¹ for accommodating a variety of heuristics for choosing resolvents. The developed heuristics and comprehensive empirical results for them are presented in this paper.

Logical Background

We assume basic familiarity with answer set semantics (see, for instance, (Baral 2003)). This section briefly introduces notations and recalls a constraint-based characterization of answer set semantics according to (Gebser *et al.* 2007c). We consider propositional (normal) logic programs over an alphabet \mathcal{P} . A *logic program* is a finite set of *rules*

$$p_0 \leftarrow p_1, \dots, p_m, \sim p_{m+1}, \dots, \sim p_n \quad (1)$$

¹<http://www.cs.uni-potsdam.de/clasp>

where $0 \leq m \leq n$ and $p_i \in \mathcal{P}$ is an *atom* for $0 \leq i \leq n$. For a rule r as in (1), let $\text{head}(r) = p_0$ be the *head* of r and $\text{body}(r) = \{p_1, \dots, p_m, \sim p_{m+1}, \dots, \sim p_n\}$ be the *body* of r . The set of atoms occurring in a logic program Π is denoted by $\text{atom}(\Pi)$, and the set of bodies in Π is $\text{body}(\Pi) = \{\text{body}(r) \mid r \in \Pi\}$. For regrouping bodies sharing the same head p , define $\text{body}(p) = \{\text{body}(r) \mid r \in \Pi, \text{head}(r) = p\}$.

For characterizing the answer sets of a program Π , we consider Boolean assignments A over domain $\text{dom}(A) = \text{atom}(\Pi) \cup \text{body}(\Pi)$. Formally, an *assignment* A is a sequence $(\sigma_1, \dots, \sigma_n)$ of (signed) *literals* σ_i of the form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in \text{dom}(A)$ and $1 \leq i \leq n$. Intuitively, $\mathbf{T}v$ expresses that v is *true* and $\mathbf{F}v$ that it is *false* in A . We denote the complement of a literal σ by $\bar{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. Furthermore, we let $A \circ B$ denote the sequence obtained by concatenating two assignments A and B . We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false propositions in A via $A^{\mathbf{T}} = \{p \in \text{dom}(A) \mid \mathbf{T}p \in A\}$ and $A^{\mathbf{F}} = \{p \in \text{dom}(A) \mid \mathbf{F}p \in A\}$. Finally, we denote the prefix of A up to a literal σ by

$$A[\sigma] = \begin{cases} (\sigma_1, \dots, \sigma_m) & \text{if } A = (\sigma_1, \dots, \sigma_m, \sigma, \dots, \sigma_n) \\ A & \text{if } \sigma \notin A. \end{cases}$$

In our context, a *nogood* (Dechter 2003) is a set $\{\sigma_1, \dots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment containing $\sigma_1, \dots, \sigma_m$. An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \text{dom}(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a *solution* for a set Δ of nogoods if $\delta \not\subseteq A$ for all $\delta \in \Delta$. Given a logic program Π , we below specify nogoods such that their solutions correspond to the answer sets of Π .

We start by describing nogoods capturing the models of the Clark's *completion* (Clark 1978) of a program Π . For $(\beta = \{p_1, \dots, p_m, \sim p_{m+1}, \dots, \sim p_n\}) \in \text{body}(\Pi)$, let

$$\Delta_\beta = \left\{ \begin{aligned} &\{\mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n, \mathbf{F}\beta\}, \\ &\{\mathbf{F}p_1, \mathbf{T}\beta\}, \dots, \{\mathbf{F}p_m, \mathbf{T}\beta\}, \\ &\{\mathbf{T}p_{m+1}, \mathbf{T}\beta\}, \dots, \{\mathbf{T}p_n, \mathbf{T}\beta\} \end{aligned} \right\}.$$

Observe that every solution for Δ_β must assign body β equivalent to the conjunction of its elements. Similarly, for an atom $p \in \text{atom}(\Pi)$, the following nogoods stipulate p to be equivalent to the disjunction of $\text{body}(p) = \{\beta_1, \dots, \beta_k\}$:

$$\Delta_p = \left\{ \begin{aligned} &\{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}p\}, \\ &\{\mathbf{T}\beta_1, \mathbf{F}p\}, \dots, \{\mathbf{T}\beta_k, \mathbf{F}p\} \end{aligned} \right\}.$$

Combining the above nogoods for Π , we get

$$\Delta_\Pi = \bigcup_{\beta \in \text{body}(\Pi)} \Delta_\beta \cup \bigcup_{p \in \text{atom}(\Pi)} \Delta_p.$$

The solutions for Δ_Π correspond one-to-one to the models of the completion of Π . If Π is *tight* (Fages 1994; Erdem & Lifschitz 2003), these models are guaranteed to match the answer sets of Π . This can be formally stated as follows.

Theorem 1 ((Gebser et al. 2007c)) *Let Π be a tight logic program. Then, $X \subseteq \text{atom}(\Pi)$ is an answer set of Π iff $X = A^{\mathbf{T}} \cap \text{atom}(\Pi)$ for a (unique) solution A for Δ_Π .*

We proceed by considering non-tight programs Π . As shown in (Lin & Zhao 2004), *loop formulas* can be added to the completion of Π to establish full correspondence to the answer sets of Π . For $U \subseteq \text{atom}(\Pi)$, let $EB_\Pi(U)$ be

$$\{\text{body}(r) \mid r \in \Pi, \text{head}(r) \in U, \text{body}(r) \cap U = \emptyset\}.$$

Observe that $EB_\Pi(U)$ contains the bodies of all rules in Π that can *externally support* (Lee 2005) an atom in U . Given $U = \{p_1, \dots, p_j\}$ and $EB_\Pi(U) = \{\beta_1, \dots, \beta_k\}$, the following nogoods capture the loop formula of U :

$$\Lambda_U = \left\{ \begin{aligned} &\{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}p_1\}, \dots, \\ &\{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}p_j\} \end{aligned} \right\}.$$

Furthermore, we define

$$\Lambda_\Pi = \bigcup_{U \subseteq \text{atom}(\Pi)} \Lambda_U.$$

By augmenting Δ_Π with Λ_Π , Theorem 1 can be extended to non-tight programs.

Theorem 2 ((Gebser et al. 2007c)) *Let Π be a logic program. Then, $X \subseteq \text{atom}(\Pi)$ is an answer set of Π iff $X = A^{\mathbf{T}} \cap \text{atom}(\Pi)$ for a (unique) solution A for $\Delta_\Pi \cup \Lambda_\Pi$.*

By virtue of Theorem 2, the nogoods in $\Delta_\Pi \cup \Lambda_\Pi$ provide us with a constraint-based characterization of the *answer sets* of Π . However, it is important to note that the size of Δ_Π is linear in $\text{atom}(\Pi) \times \text{body}(\Pi)$, while Λ_Π contains exponentially many nogoods. As shown in (Lifschitz & Razborov 2006), under current assumptions in complexity theory, the exponential number of elements in Λ_Π is inherent, that is, it cannot be reduced significantly in the worst case. Hence, ASP solvers do not determine the nogoods in Λ_Π a priori, but include mechanisms to determine them on demand. This is illustrated further in the next section.

Algorithmic Background

This section recalls the basic decision procedure of *clasp* (Gebser et al. 2007c), abstracting Conflict-Driven Clause Learning (CDCL; (Mitchell 2005)) for SAT solving from clauses, that is, Conflict-Driven Nogood Learning (CDNL).

Conflict-Driven Nogood Learning

Algorithm 1 shows our main procedure for deciding whether a program Π has some answer set. The algorithm starts with an empty assignment A and an empty set ∇ of recorded nogoods (Lines 1–2). Note that dynamic nogoods added to ∇ in Line 5 are elements of Λ_Π , while those added in Line 9 result from conflict analysis (Line 8). In addition to conflict-driven learning, the procedure performs backjumping (Lines 10–11), guided by a decision level k determined by conflict analysis. Via decision level dl , we count *decision literals*, that is, literals in A that have been heuristically selected in Line 15. The initial value of dl is 0 (Line 3), and it is incremented in Line 16 before a decision literal is added to A (Line 17). All literals in A that are not decision literals have been derived by propagation in Line 5, and we call them *implied literals*. For any literal σ in A , we write $dl(\sigma)$ to refer to the decision level of σ , that is, the value dl had when σ was added to A . After propagation, the main loop

Algorithm 1: CDNL

Input : A program Π .**Output**: An answer set of Π .

```

1  $A \leftarrow \emptyset$  // assignment over  $atom(\Pi) \cup body(\Pi)$ 
2  $\nabla \leftarrow \emptyset$  // set of (dynamic) nogoods
3  $dl \leftarrow 0$  // decision level
4 loop
5    $(A, \nabla) \leftarrow \text{PROPAGATION}(\Pi, \nabla, A)$ 
6   if  $\varepsilon \subseteq A$  for some  $\varepsilon \in \Delta_\Pi \cup \nabla$  then
7     if  $dl = 0$  then return no answer set
8      $(\delta, k) \leftarrow \text{CONFLICTANALYSIS}(\varepsilon, \Pi, \nabla, A)$ 
9      $\nabla \leftarrow \nabla \cup \{\delta\}$ 
10     $A \leftarrow A \setminus \{\sigma \in A \mid k < dl(\sigma)\}$ 
11     $dl \leftarrow k$ 
12  else if  $A^T \cup A^F = atom(\Pi) \cup body(\Pi)$  then
13    return  $A^T \cap atom(\Pi)$ 
14  else
15     $\sigma_d \leftarrow \text{SELECT}(\Pi, \nabla, A)$ 
16     $dl \leftarrow dl + 1$ 
17     $A \leftarrow A \circ (\sigma_d)$ 

```

(Lines 4–17) distinguishes three cases: a conflict detected via a violated nogood (Lines 6–11), a solution (Lines 12–13), or a heuristic selection with respect to a partial assignment (Lines 14–17). Finally, note that a conflict at decision level 0 signals that Π has no answer set (Line 7).

Propagation

Our propagation procedure, shown in Algorithm 2, derives implied literals and adds them to A . Lines 3–9 describe unit propagation (cf. (Mitchell 2005)) on $\Delta_\Pi \cup \nabla$. If a conflict is detected in Line 4, unit propagation terminates immediately (Line 5). Otherwise, in Line 6, we determine all nogoods δ that are *unit-resulting* wrt A , that is, the complement $\bar{\sigma}$ of some literal $\sigma \in \delta$ must be added to A because all other literals of δ are already true in A . If there is some unit-resulting nogood δ (Line 7), A is augmented with $\bar{\sigma}$ in Line 8. Observe that δ is chosen non-deterministically, and several distinct nogoods may imply $\bar{\sigma}$ wrt A . This non-determinism gives rise to our study of heuristics for conflict resolution, selecting a resolvent among the nogoods δ that imply $\bar{\sigma}$.

The second part of Algorithm 2 (Lines 10–14) checks for unit-resulting or violated nogoods in Λ_Π . If Π is tight (Line 10), sophisticated checks are unnecessary (cf. Theorem 1). Otherwise, we consider sets $U \subseteq atom(\Pi)$ such that $EB_\Pi(U) \subseteq A^F$, called *unfounded sets* (Van Gelder, Ross, & Schlipf 1991). An unfounded set U is determined in Line 12 by a dedicated algorithm, where $U \cap A^F = \emptyset$. If such a nonempty unfounded set U exists, each nogood $\delta \in \Lambda_U$ is either unit-resulting or violated wrt A , and an arbitrary $\delta \in \Lambda_U$ is recorded in Line 14 for triggering unit propagation. Note that all atoms in U must be falsified before another unfounded set is determined (cf. Lines 11–12). Eventually, propagation terminates in Line 13 if no nonempty unfounded set has been detected in Line 12.

Algorithm 2: PROPAGATION

Input : A program Π , a set ∇ of nogoods, and an assignment A .**Output**: An extended assignment and set of nogoods.

```

1  $U \leftarrow \emptyset$  // unfounded set
2 loop
3   repeat
4     if  $\delta \subseteq A$  for some  $\delta \in \Delta_\Pi \cup \nabla$  then
5       return  $(A, \nabla)$ 
6      $\Sigma \leftarrow \{\delta \in \Delta_\Pi \cup \nabla \mid \delta \setminus A = \{\sigma\}, \bar{\sigma} \notin A\}$ 
7     if  $\Sigma \neq \emptyset$  then let  $\sigma \in \delta \setminus A$  for some  $\delta \in \Sigma$  in
8        $A \leftarrow A \circ (\bar{\sigma})$ 
9   until  $\Sigma = \emptyset$ 
10  if  $\text{TIGHT}(\Pi)$  then return  $(A, \nabla)$ 
11   $U \leftarrow U \cup A^F$ 
12  if  $U = \emptyset$  then  $U \leftarrow \text{UNFOUNDEDSET}(\Pi, A)$ 
13  if  $U = \emptyset$  then return  $(A, \nabla)$ 
14  let  $\delta \in \Lambda_U$  in  $\nabla \leftarrow \nabla \cup \{\delta\}$ 

```

Algorithm 3: CONFLICTANALYSIS

Input : A violated nogood δ , a program Π , a set ∇ of nogoods, and an assignment A .**Output**: A derived nogood and a decision level.

```

1 loop
2   let  $\sigma \in \delta$  such that  $\delta \setminus A[\sigma] = \{\sigma\}$ 
3    $k \leftarrow \max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$ 
4   if  $k = dl(\sigma)$  then
5      $\Sigma \leftarrow \{\varepsilon \in \Delta_\Pi \cup \nabla \mid \varepsilon \setminus A[\sigma] = \{\bar{\sigma}\}\}$ 
6      $\varepsilon \leftarrow \text{SELECTANTECEDENT}(\Sigma)$ 
7      $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\bar{\sigma}\})$ 
8   else return  $(\delta, k)$ 

```

Conflict Analysis

Algorithm 3 shows our conflict analysis procedure, which is based on resolution. Given a nogood δ that is violated wrt A , we determine in Line 2 the literal $\sigma \in \delta$ added last to A . If σ is the single literal of its decision level $dl(\sigma)$ in δ (cf. Line 3), it is called a *unique implication point* (UIP; (Marques-Silva & Sakallah 1999)). Among a number of conflict resolution schemes, the *First-UIP* scheme, stopping conflict resolution as soon as the first UIP is reached, has turned out to be the most efficient and most robust strategy (Zhang *et al.* 2001). Our conflict analysis procedure follows the First-UIP scheme by performing conflict resolution only if σ is not a UIP (tested in Line 4) and, otherwise, returning δ along with the smallest decision level k at which $\bar{\sigma}$ is implied by δ after backjumping (Line 8).

Let us take a closer look at conflict resolution steps in Lines 5–7. It is important to note that, if σ is not a UIP, it cannot be the decision literal of $dl(\sigma)$. Rather, it must have been implied by some nogood $\varepsilon \in \Delta_\Pi \cup \nabla$. As a consequence, the set Σ determined in Line 5 cannot be empty, and

we call its elements *antecedents* of σ . Note that each antecedent ε contains $\bar{\sigma}$ and had been unit-resulting immediately before σ was added to A ; we thus call $\varepsilon \setminus \{\bar{\sigma}\}$ a *reason* for σ . Knowing that σ may have more than one antecedent, a non-deterministic choice among them is made in Line 6. Exactly this choice is subject to the heuristics studied below. Furthermore, as σ is the literal of δ added last to A , $\delta \setminus \{\sigma\}$ is also a reason for $\bar{\sigma}$. Since they imply complementary literals, no solution can jointly contain both reasons, viz., $\delta \setminus \{\sigma\}$ and $\varepsilon \setminus \{\bar{\sigma}\}$. Hence, combining them in Line 7 gives again a nogood violated wrt A . Finally, note that conflict resolution is guaranteed to terminate at some UIP, but different heuristic choices in Line 6 may result in different UIPs.

Implication Graphs and Conflict Graphs

To portray the matter of choosing among several distinct antecedents, we modify the notion of an implication graph (Beame, Kautz, & Sabharwal 2004). At a given state of CDNL, the *implication graph* contains a node for each literal σ in assignment A and, for a violated nogood $\delta \subseteq A$, a node $\bar{\sigma}$ is included, where σ is the literal of δ added last to A , that is, $\delta \setminus A[\sigma] = \{\sigma\}$. Furthermore, for each antecedent δ of an implied literal σ , the implication graph contains directed edges labeled with δ from all literals in the reason $\delta \setminus \{\bar{\sigma}\}$ to σ . Different from (Beame, Kautz, & Sabharwal 2004), where implication graphs reflect exactly one reason per implied literal, our implication graph thus includes all of them. If the implication graph contains both σ and $\bar{\sigma}$, we call them *conflicting literals*. Note that an implication graph contains at most one such pair $\{\sigma, \bar{\sigma}\}$, called *conflicting assignment*, because our propagation procedure in Algorithm 2 stops as soon as a nogood becomes violated (cf. Lines 4–5).

An exemplary implication graph is shown in Figure 1. Each of its nodes (except for one among the two conflicting literals) corresponds to a literal that is true in assignment

$$A = (\mathbf{Fa}, \mathbf{Fb}, \mathbf{Fp}, \mathbf{Tq}, \mathbf{Tr}, \mathbf{T}s, \mathbf{Fv}, \mathbf{Tt}, \mathbf{Fu}, \mathbf{Fw}, \mathbf{Tx}) .$$

The three decision literals in A are underlined, and all other literals are implied. For each literal σ , its decision level $dl(\sigma)$ is also provided in Figure 1 in parentheses. Every edge is labeled with at least one antecedent of its target, that is, the edges represent the following nogoods:

$$\begin{array}{ll} n_0 = \{\mathbf{Fa}, \mathbf{Tb}\} & n_1 = \{\mathbf{Tr}, \mathbf{Fs}\} \\ n_2 = \{\mathbf{T}s, \mathbf{Ft}\} & n_3 = \{\mathbf{T}s, \mathbf{Tu}\} \\ n_4 = \{\mathbf{T}s, \mathbf{Tu}\} & n_5 = \{\mathbf{Tr}, \mathbf{Tv}\} \\ n_6 = \{\mathbf{Tq}, \mathbf{Fv}, \mathbf{Tu}\} & n_7 = \{\mathbf{Tt}, \mathbf{Fu}, \mathbf{Fx}\} \\ n_8 = \{\mathbf{Fp}, \mathbf{Tt}, \mathbf{Fx}\} & n_9 = \{\mathbf{Fw}, \mathbf{Tx}\} . \end{array}$$

Furthermore, nogood $\{\mathbf{Ta}\}$ is unit-resulting wrt the empty assignment, thus, implied literal \mathbf{Fa} (whose decision level is 0) does not have any incoming edge. Observe that the implication graph contains conflicting assignment $\{\mathbf{Tx}, \mathbf{Fx}\}$, where \mathbf{Tx} has been implied by nogood n_7 and likewise by n_8 . It is also the last literal in A belonging to violated nogood n_9 , so that its complement \mathbf{Fx} is the second conflicting literal in the implication graph. Besides \mathbf{Tx} , literal \mathbf{Fw} has multiple antecedents, namely, n_4 and n_6 , which can be read off the labels of the incoming edges of \mathbf{Fw} .

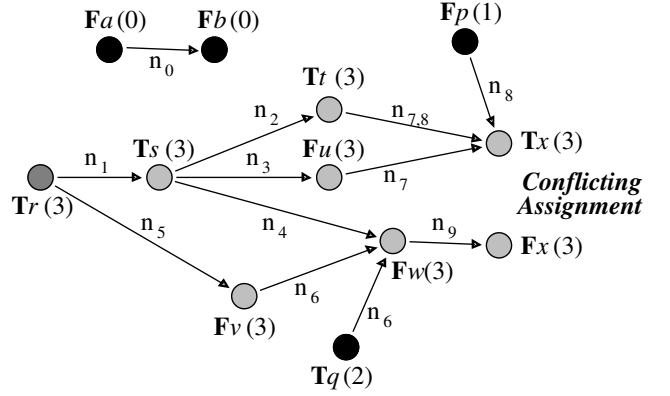


Figure 1: An exemplary implication graph containing a conflicting assignment.

The conflict resolution done in Algorithm 3, in particular, the heuristic choice of antecedents in Line 6, can now be viewed as an iterative projection of the implication graph. In fact, if an implied literal has incoming edges with distinct labels, all edges with a particular label are taken into account, while the edges with different labels only are dropped. This observation motivates the following definition: a subgraph of an implication graph is a *conflict graph* if it contains a conflicting assignment and, for each implied literal σ in the subgraph, the set of predecessors of σ is a reason for σ . Note that this definition allows us to drop all literals that do not have a path to any conflicting literal, such as \mathbf{Fa} and \mathbf{Fb} in Figure 1. Furthermore, the requirement that the predecessors of an implied literal form a reason corresponds to the selection of an antecedent, where only the incoming edges with a particular label are traced via conflict resolution.

The next definition accounts for a particularity of ASP solving related to unfounded set handling: a conflict graph is *level-aware* if each conflicting literal σ has some predecessor ρ such that $dl(\rho) = dl(\sigma)$. In fact, propagation in Algorithm 2 is limited to falsifying unfounded atoms, thus, unit propagation on nogoods in Λ_Π is performed only partially and may miss implied literals corresponding to external bodies (cf. (Gebser *et al.* 2007c)). If a conflict graph is not level-aware, the violated nogood δ provided as input to Algorithm 3 already contains a UIP, thus, δ itself is returned without performing any conflict resolution in-between. Given that we are interested in conflict resolution, we below consider level-aware conflict graphs only.

Finally, we characterize nogoods derived by Algorithm 3 by cuts in conflict graphs (cf. (Zhang *et al.* 2001; Beame, Kautz, & Sabharwal 2004)). A *conflict cut* in a conflict graph is a bipartition of the nodes such that all decision literals belong to one side, called *reason side*, and the conflicting assignment is contained in the other side, called *conflict side*. The set of nodes on the reason side that have some edge into the conflict side form the *conflict nogood* associated with a particular conflict cut. For illustration, a First-New-Cut (Beame, Kautz, & Sabharwal 2004) is shown in Figure 2. For the underlying conflict graph, we can choose among the

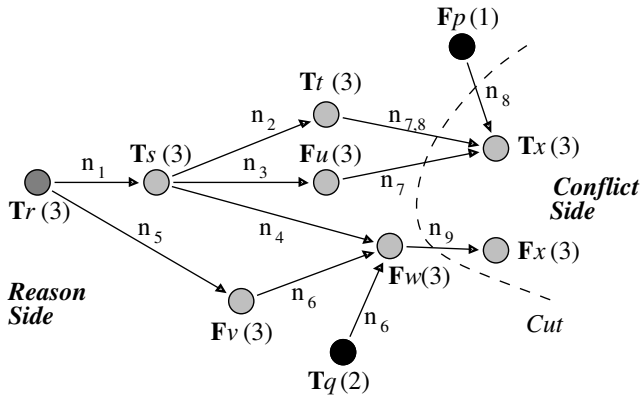


Figure 2: The implication graph with a First-New-Cut.

incoming edges of Tx whether to include the edges labeled with n_7 or the ones labeled with n_8 . With n_7 , we get conflict nogood $\{Tt, Fu, Fw\}$, while n_8 yields $\{Fp, Tt, Fw\}$.

Different conflict cuts correspond to different resolution schemes, where we are particularly interested in the First-UIP scheme. Given a conflict graph and conflicting assignment $\{\sigma, \bar{\sigma}\}$, a UIP σ_{UIP} can be identified as a node such that all paths from σ_d , the decision literal of decision level $dl(\sigma) = dl(\bar{\sigma})$, to either σ or $\bar{\sigma}$ go through σ_{UIP} (cf. (Zhang et al. 2001)). In view of this alternative definition of a UIP, it becomes even more obvious than before that σ_d is indeed a UIP, also called the Last-UIP. In contrast, a literal σ_{UIP} is the First-UIP if it is the UIP “closest” to the conflicting literals, that is, if no other UIP is reachable from σ_{UIP} . The *First-UIP-Cut* is then given by the conflict cut that has all literals lying on some path from the First-UIP to a conflicting literal, except for the First-UIP itself, on the conflict side and all other literals (including the First-UIP) on the reason side. The *First-UIP-Nogood*, that is, the conflict nogood associated with the First-UIP-Cut, is exactly the nogood derived by conflict resolution in Algorithm 3 when antecedents that contribute edges to the conflict graph are selected for conflict resolution. Also note that the First-UIP-Cut for a conflict graph is unique, thus, by projecting an implication graph to a conflict graph, we implicitly fix the First-UIP-Nogood. With this in mind, the next section deals with heuristics for extracting conflict graphs from implication graphs.

Heuristics

In this section, we propose several heuristics for conflict resolution striving for different goals.

Recording Short Nogoods

Under the assumption that short nogoods prune larger portions of the search space than longer ones, a First-UIP-Nogood looks the more attractive the less literals it contains. In addition, unit propagation on shorter nogoods is usually faster and might even be enabled to use particularly optimized data structures, for instance, specialized to binary or ternary nogoods (Ryan 2004). As noticed in (Mahajan, Fu,

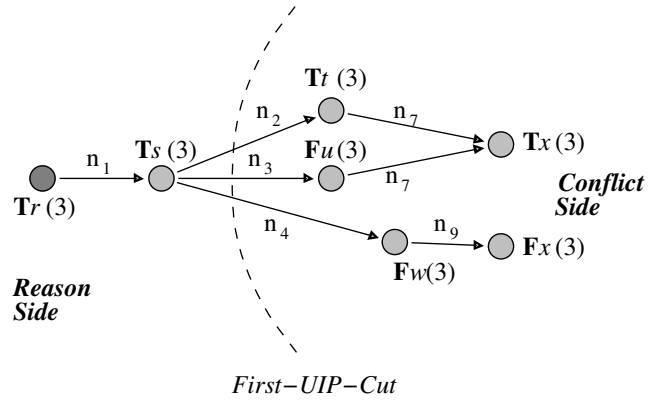


Figure 3: A First-UIP-Cut obtained with H_{short} .

& Malik 2005), a conflict nogood stays short when the resolvents are short, when the number of resolvents is small, or when the resolvents have many literals in common. In the SAT area, it has been observed that preferring short nogoods in conflict resolution may lead to resolution sequences involving mostly binary and ternary nogoods, so that derived conflict nogoods are not much longer than the originally violated nogoods (Mitchell 2005). Our first heuristics, H_{short} , thus selects an antecedent containing the smallest number of literals among the available antecedents of a literal. Given the same implication graph as in Figure 1 and 2, H_{short} may yield the conflict graph shown in Figure 3 by preferring antecedent n_7 of Tx over n_8 and antecedent n_4 of Fw over n_6 during conflict resolution. The corresponding First-UIP-Nogood, $\{Ts\}$, is indeed short and enables CDNL to after backjumping derive Fs by unit propagation at decision level 0. However, the antecedents n_7 and n_8 of Tx are of the same size, thus, H_{short} may likewise pick n_8 , in which case the First-UIP-Cut in Figure 4 is obtained. The corresponding First-UIP-Nogood, $\{Fp, Ts\}$, is longer. Nonetheless, our experiments below empirically confirm that H_{short} tends to reduce the size of First-UIP-Nogoods. But before, we describe further heuristics focusing also on other aspects.

Performing Long Backjumps

By backjumping, CDNL may skip the exhaustive exploration of regions of the search space, possibly escaping sparse regions not containing any solution. Thus, it seems reasonable to aim at First-UIP-Nogoods such that their literals belong to small decision levels, as they are the determining factor for the lengths of backjumps. Our second heuristics, H_{lex} , thus uses a lexicographic order to rank antecedents according to the decision levels of their literals. Given an antecedent δ of a literal σ , we arrange the literals in the reason $\delta \setminus \{\bar{\sigma}\}$ for σ in descending order of their decision levels. The so obtained sequence $(\sigma_1, \dots, \sigma_m)$, where $\delta \setminus \{\bar{\sigma}\} = \{\sigma_1, \dots, \sigma_m\}$, induces a descending list $levels(\delta) = (dl(\sigma_1), \dots, dl(\sigma_m))$ of decision levels. An antecedent δ is then considered to be smaller than another antecedent ε , viz., $\delta < \varepsilon$, if the first element that differs in $levels(\delta)$ and $levels(\varepsilon)$ is smaller in $levels(\delta)$ or if $levels(\delta)$

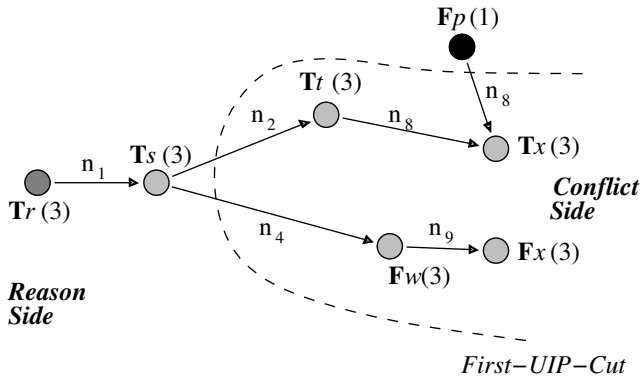


Figure 4: A First-UIP-Cut obtained with H_{lex} .

is a prefix of $levels(\varepsilon)$ and shorter than $levels(\varepsilon)$. Due to the last condition, H_{lex} also prefers an antecedent δ that is shorter than ε , provided that literals of the same decision levels as in δ are also found in ε . Reconsidering the implication graph in Figure 1 and 2, we obtain $levels(n_8) = (3, 1) < (3, 3) = levels(n_7)$ for antecedents n_7 and n_8 of Tx , and we have $levels(n_4) = (3) < (3, 2) = levels(n_6)$ for antecedents n_4 and n_6 of Fw . By selecting antecedents that are lexicographically smallest, H_{lex} leads us to the conflict graph shown in Figure 4. In this example, the corresponding First-UIP-Nogood, $\{Fp, Ts\}$, is weaker than $\{Ts\}$, which may be obtained with H_{short} (cf. Figure 3).

Given that lexicographic comparisons are computationally expensive, we also consider a lightweight variant of ranking antecedents according to decision levels. Our third heuristics, H_{avg} , prefers an antecedent δ over ε if the average of $levels(\delta)$ is smaller than the average of $levels(\varepsilon)$. In our example, we get $avg[levels(n_8)] = avg(3, 1) = 2 < 3 = avg(3, 3) = avg[levels(n_7)]$ and $avg[levels(n_6)] = avg(3, 2) = 2.5 < 3 = avg(3) = avg[levels(n_4)]$, yielding the conflict graph shown in Figure 5. Unfortunately, the corresponding First-UIP-Nogood, $\{Fp, Tq, Tr\}$, does not match the goal of H_{avg} as backjumping only returns to decision level 2, where Tr is then flipped to Fr . Note that this behavior is similar to chronological backtracking, which can be regarded as the most trivial form of backjumping.

Shortening Conflict Resolution

Our fourth heuristics, H_{res} , aims at speeding up conflict resolution itself by shortening resolution sequences. In order to earlier encounter a UIP, H_{res} prefers antecedents such that the number of literals at the current decision level dl is smallest. In our running example, H_{res} prefers n_8 over n_7 as it contains fewer literals whose decision level is 3. However, antecedents n_4 and n_6 of Fw are indifferent, thus, H_{res} may yield either one of the conflict graphs in Figure 4 and 5.

Search Space Pruning

The heuristics presented above rank antecedents merely by structural properties, thus disregarding their contribution in the past to solving the actual problem. The latter is estimated by nogood deletion heuristics of SAT solvers (Goldberg &

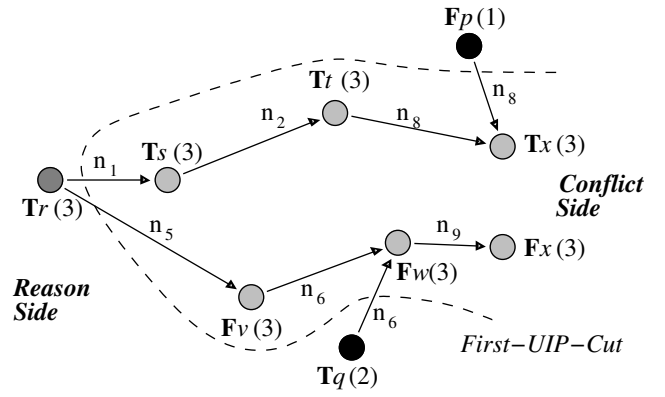


Figure 5: A First-UIP-Cut obtained with H_{avg} .

Novikov 2002; Mahajan, Fu, & Malik 2005), and *clasp* also maintains activity scores for nogoods (Gebser *et al.* 2007a). Our fifth heuristics, H_{active} , makes use of them and ranks antecedents according to their activities.

Finally, we investigate a heuristics, H_{prop} , that stores (and prefers) the smallest decision level at which a nogood has ever been unit-resulting. The intuition underlying H_{prop} is that the number of implied literals at small decision levels can be viewed as a measure for the progress of CDNL, in particular, as attesting unsatisfiability requires a conflict at decision level 0. Thus, it might be a good idea to prefer nogoods that gave rise to implications at small decision levels.

Experiments

For their empirical assessment, we have implemented the heuristics proposed above in a prototypical extension of our ASP solver *clasp* version 1.0.2. (Even though there are newer versions of *clasp*, a common testbed, omitting some optimizations, is sufficient for a representative comparison.) Note that *clasp* (Gebser *et al.* 2007a) incorporates various advanced Boolean constraint solving techniques, e.g.:

- lookback-based decision heuristics (Goldberg & Novikov 2002),
- restart and nogood deletion policies (Eén & Sörensson 2003),
- watched literals for unit propagation on “long” nogoods (Moskewicz *et al.* 2001),
- dedicated treatment of binary and ternary nogoods (Ryan 2004), and
- early conflict detection (Mahajan, Fu, & Malik 2005).

Due to this variety, the solving process of *clasp* is a complex interplay of different features. Thus, it is almost impossible to observe the impact of a certain feature, such as our conflict resolution heuristics, in isolation. However, we below use a considerable number of benchmark classes with different characteristics and shuffled instances, so that noise effects should be compensated at large.

For accommodating conflict resolution heuristics considering several antecedents per literal, the low-level implementation of *clasp* had to be modified. These modifications

are less optimized than the original implementation, so that our prototype incurs some disadvantages in raw speed that can potentially be reduced by optimizing the implementation. However, for comparison, we include unmodified *clasp* version 1.0.2, not applying any particular heuristics in conflict resolution. Given that unit propagation in *clasp* privileges binary and ternary nogoods, they are more likely to be used as antecedents than longer nogoods, as original *clasp* simply stores the first antecedent it encounters and ignores others. In view of this, unit propagation of original *clasp* leads conflict resolution into the same direction as H_{short} , though in a less exact way. The next table summarizes all *clasp* variants and conflict resolution heuristics under consideration, denoting the unmodified version simply by *clasp*:

Label	Heuristics	Goal
<i>clasp</i>	—	speeding up unit propagation
<i>clasp_{short}</i>	H_{short}	recording short nogoods
<i>clasp_{lex}</i>	H_{lex}	performing long backjumps
<i>clasp_{avg}</i>	H_{avg}	performing long backjumps
<i>clasp_{res}</i>	H_{res}	shortening conflict resolution
<i>clasp_{active}</i>	H_{active}	search space pruning
<i>clasp_{prop}</i>	H_{prop}	search space pruning

Note that all *clasp* variants perform early conflict detection, that is, they encounter a unique conflicting assignment before beginning with conflict resolution. Furthermore, all of them perform conflict resolution according to the First-UIP scheme. Thus, we do not explore the first two among the three degrees of freedom mentioned in the introductory section and concentrate fully on the choice of resolvents.

We conducted experiments on the benchmarks used in categories *SCore* and *SLparse* of the first ASP system competition (Gebser *et al.* 2007d). Tables 1–4 group benchmark instances by their classes, viz., Classes 1–11. Via superscripts ^s and ^r in the first column, we indicate whether the n instances belonging to a class are structured (e.g., 15-Puzzle) or randomly generated (e.g., BlockedN-Queens). We omit classifying Factoring, which is a worst-case problem where an efficient algorithm would yield a cryptographic attack. Furthermore, Tables 1–4 show results for computing one answer set or deciding that an instance has no answer set. For each benchmark instance, we performed five runs on different shuffles, resulting in $5n$ runs per benchmark class. All experiments were run on a 3.4GHz PC under Linux; each run was limited to 600s time and 1GB RAM. Note that, in Tables 1–3, we consider only the instances on which runs were completed by all considered *clasp* variants.

Table 1 shows the average lengths of First-UIP-Nogoods for the heuristics aiming at short nogoods, implemented by *clasp_{short}* and *clasp_{lex}*, among which the latter uses the lengths of antecedents as a tie breaker. For comparison, we also include original *clasp*. On most benchmark classes, we observe that *clasp_{short}* as well as *clasp_{lex}* tend to reduce the lengths of First-UIP-Nogoods, up to 14 percent shorter than the ones of *clasp* on BlockedN-Queens. But there remains only a slight reduction of about 6 percent shorter First-UIP-Nogoods of *clasp_{lex}* in the summary of all benchmark classes (weighted equally). We also observe that *clasp_{short}*, more straightly preferring short antecedents than *clasp_{lex}*,

No.	Class	n	<i>clasp_{short}</i>	<i>clasp_{lex}</i>	<i>clasp</i>
1 ^s	15-Puzzle	10	22.33	22.35	23.03
2 ^r	BlockedN-Queens	7	27.32	28.23	31.85
3 ^s	EqTest	5	172.12	178.27	189.12
4	Factoring	5	134.95	130.67	141.34
5 ^s	HamiltonianPath	14	12.96	11.73	12.04
6 ^r	RandomNonTight	14	31.82	32.07	32.74
7 ^r	BoundedSpanningTree	5	35.06	36.68	33.95
8 ^s	Solitaire	4	24.55	22.02	25.03
9 ^s	Su-Doku	3	16.22	15.09	13.99
10 ^s	TowersOfHanoi	5	52.89	52.31	58.29
11 ^r	TravelingSalesperson	5	101.37	90.35	99.26
Average First-UIP-Nogood Length			45.15	44.46	47.21

Table 1: Average lengths of First-UIP-Nogoods per conflict.

No.	Class	n	<i>clasp_{avg}</i>	<i>clasp_{lex}</i>	<i>clasp</i>
1 ^s	15-Puzzle	10	2.12	2.14	2.10
2 ^r	BlockedN-Queens	7	1.07	1.08	1.07
3 ^s	EqTest	5	1.03	1.04	1.03
4	Factoring	5	1.20	1.21	1.20
5 ^s	HamiltonianPath	14	2.53	2.58	2.62
6 ^r	RandomNonTight	14	1.15	1.16	1.15
7 ^r	BoundedSpanningTree	5	3.12	3.47	3.06
8 ^s	Solitaire	4	3.34	3.28	2.92
9 ^s	Su-Doku	3	2.55	3.01	2.76
10 ^s	TowersOfHanoi	5	1.46	1.46	1.40
11 ^r	TravelingSalesperson	5	1.27	1.51	1.43
Average Backjump Length			1.89	1.99	1.89

Table 2: Average backjump lengths per conflict.

does not reduce First-UIP-Nogood lengths any further. Interestingly, there is no clear distinction between structured and randomly generated instances, neither regarding magnitudes nor reduction rates of First-UIP-Nogood lengths.

Table 2 shows the average backjump lengths in terms of decision levels for the *clasp* variants aiming at long backjumps, viz., *clasp_{avg}* and *clasp_{lex}*. We note that average backjump lengths of more than 2 decision levels indicate structured instances, except for BoundedSpanningTree. Regarding the increase of backjump lengths, *clasp_{avg}* does not exhibit significant improvements, and the polarity of differences to original *clasp* varies. Only the more sophisticated heuristics of *clasp_{lex}* almost consistently leads to increased backjump lengths (except for HamiltonianPath), but the amounts of improvements are rather small.

Table 3 shows the average numbers of conflict resolution steps for *clasp_{res}* and *clasp_{lex}*, among which the former particularly aims at their reduction. Somewhat surprisingly, *clasp_{res}* in all performs more conflict resolution steps even than original *clasp*, while *clasp_{lex}* almost consistently exhibits a reduction of conflict resolution steps (except for Su-Doku). This negative result for *clasp_{res}* suggests that trimming conflict resolution regardless of its outcome is not advisable. The quality of recorded nogoods certainly is a key factor for the performance of conflict-driven learning solvers for ASP and SAT, thus, shallow savings in their retrieval are not worth it and might even be counterproductive globally.

No.	Class	n	$clasp_{short}$	$clasp_{lex}$	$clasp_{avg}$	$clasp_{res}$	$clasp_{active}$	$clasp_{prop}$	$clasp$
1 ^s	15-Puzzle	10	195.00	203.96	203.54	248.00	261.44	226.96	241.18
			0.13	0.14	0.14	0.15	0.16	0.15	0.14
2 ^r	BlockedN-Queens	7	27289.06	26989.57	28176.00	27553.63	30240.71	29119.60	28588.34
			116.87 (24)	122.04 (21)	39.70 (27)	86.24 (24)	138.01 (22)	68.10 (25)	24.52 (22)
3 ^s	EqTest	5	62430.92	62648.96	59330.52	62705.00	62374.84	63303.44	62290.76
			19.47	21.66	19.41	19.98	20.03	21.30	15.66
4	Factoring	5	15468.44	14838.64	14985.72	16016.56	16365.52	15404.64	16920.68
			6.30	5.85	6.27	6.55	6.36	6.36	5.11
5 ^s	HamiltonianPath	14	703.70	683.29	653.19	564.83	764.16	694.33	650.70
			0.05	0.05	0.05	0.04	0.06	0.05	0.05
6 ^r	RandomNonTight	14	427031.71	411024.73	402846.21	429955.23	423332.74	405476.81	406007.41
			53.85	55.17	51.53	54.92	53.33	52.78	41.79
7 ^r	BoundedSpanningTree	5	879.92	640.88	801.76	634.96	662.22	940.92	949.84
			4.51	4.37	4.38	4.36	4.27	4.98	4.42
8 ^s	Solitaire	4	193.85	145.85	103.40	134.75	103.40	95.90	134.00
			66.14 (2)	0.22 (5)	30.81 (4)	0.22 (5)	0.21 (5)	0.21 (5)	0.23 (4)
9 ^s	Su-Doku	3	123.40	127.80	164.60	111.93	108.67	119.87	123.93
			18.89	19.85	19.75	19.10	19.39	19.77	19.96
10 ^s	TowersOfHanoi	5	145064.20	124222.96	71220.52	140386.64	97411.80	134192.96	133760.48
			62.43	46.69	21.86	52.19	32.76	47.63	37.60
11 ^r	TravelingSalesperson	5	2512.20	1018.80	3243.16	2535.40	1334.32	2500.16	947.56
			34.06	21.63	42.22	36.77	25.70	34.42	20.89
Average Number of Conflicts			56824.37	53545.45	48477.39	56737.24	52748.20	54339.63	54217.91
Average Time (Sum Timeouts)			31.89 (26)	24.81 (26)	19.68 (31)	23.38 (29)	25.02 (27)	21.31 (30)	14.20 (26)
Average Penalized Time			49.25	46.75	45.28	48.05	47.12	47.14	37.27

Table 4: Average numbers of conflicts and runtimes.

No.	Class	n	$clasp_{res}$	$clasp_{lex}$	$clasp$
1 ^s	15-Puzzle	10	102.95	103.45	103.77
2 ^r	BlockedN-Queens	7	18.17	17.61	17.74
3 ^s	EqTest	5	86.94	84.78	85.76
4	Factoring	5	325.54	290.36	296.07
5 ^s	HamiltonianPath	14	11.87	12.03	12.14
6 ^r	RandomNonTight	14	16.41	16.47	32.74
7 ^r	BoundedSpanningTree	5	20.11	20.27	20.66
8 ^s	Solitaire	4	79.05	67.70	79.89
9 ^s	Su-Doku	3	21.48	20.86	19.73
10 ^s	TowersOfHanoi	5	41.60	40.36	42.69
11 ^r	TravelingSalesperson	5	141.68	96.06	122.98
Average Number of Resolution Steps			78.71	70.00	75.83

Table 3: Average numbers of resolution steps per conflict.

Finally, Table 4 provides average numbers of conflicts and average runtimes in seconds for all *clasp* variants. For each benchmark class, the first line provides the average numbers of conflicts encountered on instances where runs were completed by all *clasp* variants, while the second line gives the average times of completed runs and numbers of timeouts in parentheses. (Recall that all *clasp* variants were run on $5n$ shuffles of the n instances per class, leading to more than n timeouts on BlockedN-Queens and, with some *clasp* variants, also on Solitaire.) At the bottom of Table 4, we summarize average numbers of conflicts and average runtimes over all benchmark classes (weighted equally). Note that the last but one line provides the sums of timeouts in parentheses, while the last line penalizes timeouts with max-

imum time, viz., 600 seconds. As mentioned above, original *clasp* is highly optimized and does not suffer from the overhead incurred by the extended infrastructure for applying heuristics in conflict resolution. As a consequence, we observe that original *clasp* outperforms its variants on most benchmark classes as regards runtime. Among the variants of *clasp*, *clasp_{avg}* in all exhibits the best average number of conflicts and runtime. However, it also times out most often and behaves unstable, as the poor performance on Classes 2 and 11 shows. In contrast, *clasp_{short}* and *clasp_{lex}* lead to fewest timeouts (in fact, as many timeouts as *clasp*), and *clasp_{lex}* encounters fewer conflicts than *clasp_{short}*. Variant *clasp_{active}*, preferring “critical” antecedents, exhibits a comparable performance, while *clasp_{res}* and *clasp_{prop}* yield more timeouts and also encounter relatively many conflicts. Overall, we notice that some *clasp* variants perform reasonably well, but without significantly decreasing the number of conflicts in comparison to original *clasp*. As there is no clear winner among our *clasp* variants, unfortunately, they do not suggest any “universal” conflict resolution heuristics.

Discussion

We have proposed a number of heuristics for conflict resolution and conducted a systematic empirical study in the context of our ASP solver *clasp*. However, it is too early to conclude any dominant approach or to make general recommendations. As has also been noted in (Mitchell 2005), conflict resolution strategies are almost certainly important but have received little attention in the literature so far. In fact, dedicated approaches in the SAT area (Ryan 2004; Mahajan, Fu,

& Malik 2005) merely aim at reducing the size of recorded nogoods. Though this might work reasonably well in practice, it is unsatisfactory when compared to sophisticated decision heuristics (Goldberg & Novikov 2002; Ryan 2004; Mahajan, Fu, & Malik 2005; Dershowitz, Hanna, & Nadel 2005) resulting from more profound considerations. We thus believe that heuristics in conflict resolution deserve further attention. Future lines of research may include developing more sophisticated scoring mechanisms than the ones proposed here, combining several scoring criteria, or even determining and possibly recording multiple reasons for a conflict (corresponding to different conflict graphs). Any future improvements in these directions may significantly boost the state-of-the-art in both ASP and SAT solving.

References

- Baral, C.; Brewka, G.; and Schlipf, J., eds. 2007. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Springer-Verlag.
- Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
- Bayardo, R., and Schrag, R. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, 203–208. AAAI Press/MIT Press.
- Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research* 22:319–351.
- Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*, 293–322. Plenum Press.
- Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.
- Dershowitz, N.; Hanna, Z.; and Nadel, A. 2005. A clause-based heuristic for SAT solvers. In Bacchus, F., and Walsh, T., eds., *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 46–60. Springer-Verlag.
- Eén, N., and Sörensson, N. 2003. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, 502–518.
- Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming* 3(4-5):499–518.
- Fages, F. 1994. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science* 1:51–60.
- Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007a. clasp: A conflict-driven answer set solver. In Baral et al. (2007), 260–265.
- Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007b. Conflict-driven answer set enumeration. In Baral et al. (2007), 136–148.
- Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007c. Conflict-driven answer set solving. In Veloso, M., ed., *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, 386–392. AAAI Press/MIT Press.
- Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczyński, M. 2007d. The first answer set programming system competition. In Baral et al. (2007), 3–17.
- Giunchiglia, E.; Lierler, Y.; and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning* 36(4):345–377.
- Goldberg, E., and Novikov, Y. 2002. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*, 142–149. IEEE Press.
- Lee, J. 2005. A model-theoretic counterpart of loop formulas. In Kaelbling, L., and Saffioti, A., eds., *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, 503–508. Professional Book Center.
- Lifschitz, V., and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 27(2):261–268.
- Lin, F., and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence* 157(1-2):115–137.
- Mahajan, Y.; Fu, Z.; and Malik, S. 2005. Zchaff2004: An efficient SAT solver. In Hoos, H., and Mitchell, D., eds., *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 360–375. Springer-Verlag.
- Marques-Silva, J., and Sakallah, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5):506–521.
- Mitchell, D. 2005. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science* 85:112–133.
- Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, 530–535. ACM Press.
- Ryan, L. 2004. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University.
- Van Gelder, A.; Ross, K.; and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650.
- Ward, J., and Schlipf, J. 2004. Answer set programming with clause learning. In Lifschitz, V., and Niemelä, I., eds., *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, 302–313. Springer-Verlag.
- Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, 279–285.